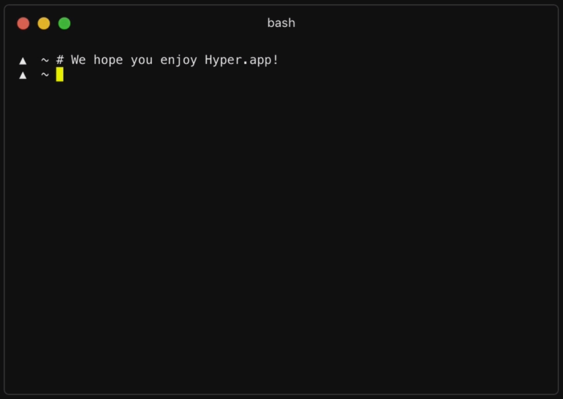
▲ Changelog GitHub License: **MIT** Download

Hyper.app

JS/HTML/CSS Terminal


bash

```
▲  ~ # We hope you enjoy Hyper.app!
▲  ~ 
```

Installation

Download for macOS and linux (ubuntu/debian). Windows build is coming very soon!

Project goals

The goal of the project is to create a beautiful and extensible experience for command-line interface users, built on open web standards.

In the beginning, our focus will be primarily around speed, stability and the development of the correct API for extension authors.

In the future, we anticipate the community will come up with innovative additions to enhance what could be the simplest, most powerful and well-tested interface for productivity.

Extensions

Extensions are available on npm. We encourage everyone to include `hyper` in the `keywords` field in `package.json`.

```
$ npm search hyper
```

Then edit `~/.hyper.js` and add it to `plugins`

```
module.exports = {

  config: { /*... */ },

  plugins: [
    "hyperpower"
  ]

};
```

`Hyper` will show a notification when your modules are installed to `~/.hyper_plugins`.

Configuration

The `config` object seen above in `~/.hyper.js` admits the following

Property	Default	Description
"fontSize"	12	The default size in pixels for the terminal

Property	Default	Description
"fontFamily"	"Menlo, DejaVu Sans Mono, Lucida Console, monospace"	The font family to use with optional fallbacks
"cursorColor"	"#F81CE5"	The color of the caret in the terminal
"cursorShape"	"BLOCK"	The shape of the caret in the terminal. Available options are: 'BEAM', 'UNDERLINE', 'BLOCK'
"foregroundColor"	"#fff"	The color of the main text of the terminal
"backgroundColor"	"#000"	The color of the window and main terminal backgroun
"borderColor"	"#333"	The color of the main window border and tab bar
"css"	""	Custom CSS to include in the main window
"termCSS"	""	Custom CSS to include in the terminal window
"padding"	"12px 14px"	CSS padding values for the space around each term
"colors"	{ black: "#000000", red: "#ff0000", ... }	A list of overrides for the color palette. The name of the keys represent the "ANSI 16", which can all seen in the default config.
"shell"	""	A path to a custom shell to run when Hyper starts a new session
"npmRegistry"	npm get registry	Override the npm registry URL to use when installin plugins
"windowSize"	null	The default width/height in pixels of a new window e.g. [540, 380]
"clearSelection"	false	If true, selected text will automatically be copied to the clipboard

Extensions API

Extensions are universal Node.js modules loaded by both Electron and the renderer process.

The extension system is designed around composition of the APIs we use to build the terminal: `React` components and `Redux` actions.

Instead of exposing a custom API method or parameter for every possible customization point, we allow you to intercept and compose every bit of functionality!

The only knowledge that is therefore required to successfully extend `Hyper` is that of its underlying open source libraries.

Your module has to expose at least one of these methods:.

Method	Invoked from	Description
`onApp`	Electron	Invoked when the app first loads. If a plugin reloads, it's invoked again with the existing app. Parameters: `app` — The electron app.
`onWindow`	Electron	Invoked when each window is created. If a plugin reloads, it' invoked again with the existing windows. Parameters: `window` — An electron `BrowserWindow`.
`onUnload`	Electron	Invoked when a plugin is removed by the user. Parameters: `app` — The electron app.
`decorateConfig`	Electron / Renderer	v0.5.0+. Allows you to decorate the user's configuration. Useful for themeing or custom parameters for your plugin. Parameters: `config` — The `config` object
`decorateEnv`	Electron	v0.7.0+. Allows you to decorate the user's environment by returning a modified environment object. Parameters: `environment` — The `environment` object
`decorateMenu`	Electron	Invoked with the Electron's `Menu` template. If a plugin reloads, it's called again and the menu is refreshed. Parameters: `menu` — The menu template object

Method	Invoked from	Description
`onRendererLoad`	Renderer	Invoked when a plugin is first loaded or subsequently reloade in each window. Parameters: `window` — The window object
`onRendererUnload`	Renderer	Invoked when a plugin is being unloaded. Parameters: `window` — The window object
`middleware`	Renderer	A custom Redux middleware that can intercept any action. Subsequently we invoke the `thunk` middleware, which means yo middleware can `next` thunks.
`reduceUI` `reduceSessions`	Renderer	A custom reducer for the `ui` state shape. `state` — The Redux state object `action` — The action object
`getTermProps`	Renderer	Passes down props from `<Terms>` to the `<Term>` component. Must return the composed props object. `uid` — Term uid `parentProps` — Props form the parent component. `props` — The existing properties that will be passed to the component.
`getTabsProps`	Renderer	Passes down props to `<Tabs>` to the `<Header>` component. Mu return the composed props object. `parentProps` — Props form the parent component. `props` — The existing properties that will be passed to the component.

Method	Invoked from	Description
`getTabProps`	Renderer	Passes down props to `<Tab>` to the `<Tabs>` component. Must return the composed props object.

`uid`	Tab / Term uid
`parentProps`	Props form the parent component.
`props`	The existing properties that will be passed to the component.

Method	Invoked from	Description
`mapHyperTermState` `mapTermsState` `mapHeaderState` `mapNotificationsState`	Renderer	A custom mapper for the state properties that container components receive. Note that for children components to get these properties, you have to pass them down using the corresponding methods (like `getTermProps`). Must return an extended object of the map passsed.

`state`	The `Redux` global state
`map`	The existing map of properties that will be passed ot the component.

Method	Invoked from	Description
`mapHyperTermDispatch` `mapTermsDispatch` `mapHeaderDispatch` `mapNotificationsDispatch`	Renderer	A custom mapper for the dispatch properties. Must return an extended object of the map passsed.

`dispatch`	The Redux dispatch function
`map`	The existing map of properties that will be passed ot the component.

Method	Invoked from	Description
`decorateHyperTerm` `decorateTerms` `decorateTerm` `decorateHeader` `decorateTabs` `decorateTab`	Renderer	Invoked with the `React` `Component` to decorate. Must return Higher Order Component. Parameters:

`HyperTerm`	The `React` `Component` constructor.
`env`	A collection of useful module references for building components. See below

Module loading

The user can hot-load and hot-reload plugins by pressing Command + R (refresh). Please strive to make plugins that don't require a complete restart of the application to work.

In the future we might do this automatically.

When developing, you can add your plugin to `~/.hyper_plugins/local` and then specify it under the `localPlugins` array in `~/.hyper.js`. We load new plugins:

- Periodically (every few hours)
- When changes are made to the configuration file (`plugins` or `localPlugins`)
- When the user clicks Plugins > Update all now

The process of reloading involves

- Running `npm prune` and `npm install` in `~/.hyper_plugins`.
- Pruning the `require.cache` in both electron and the renderer process
- Invoking `on*` methods on the existing instances and re-rendering components with the fresh decorations in place.``

Note: on the main process, plugins are registered as soon as possible (we fire `onLoad`). On the browser, it's up to the user to trigger their load by pressing command+R. We put the user in control of the loading in this way to prevent them from losing critical work by extensions that reset state or don't preserve it correctly.

Decorating components

We give you the ability to provide a higher order component for every piece of the `Hyper` UI. Its structure is as follows:

```
<HyperTerm>
  <Header>
    <Tabs>
      <Tab /> ...
    </Tabs>
  </Header>
  <Terms>
    <Term /> ...
  </Terms>
  <Notifications>
    <Notification /> ...
  </Notifications>
</HyperTerm>
```

All the `decorate*` methods receive the following references in an object passed as the second parameter:

`React`	The entire React namespace.
`notify`	A helper function that shows a desktop notification. The first parameter is the title a the second is the optional body of the notification.
`Notification`	The `Notification` component in case you want to re-use it.

All the components accept the following two properties to extend their markup:

\`customChildren\`	An array of \`Element\` or a single \`Element\` to insert at the bottom of the component.
\`customChildrenBefore\`	The same as \`customChildren\`, but inserted as the first child element(s) of the component.

We encourage you to maintain compatibility with other decorators. Since many can be set, don't assume that yours is the only one. For example, if you're passing children, compose potential existing values:

```
render () {
  const customChildren = Array.from(this.props.customChildren)
    .concat(<p>My new child</p>);
  return <Tab {...this.props} customChidren={customChildren} />
}
```

Actions and Effects

All the Redux actions are available for you to handle through your middleware and reducers. For an example, refer to the Hyperpower reference plugin.

Side effects occur in two fundamental forms:

- Some actions dispatch other actions based on state.
- Some actions do async work by communicating over the RPC channel to the main process

In all cases, the side effect is passed as the \`effect\` key in the action and later handled by our middleware.

This means that you can override, compose or completely eliminate effects! In other words, this is how you can change the default functionality or behavior of the app.

As an example, consider the action we use to increase the font size when you press \`Command+=\`:

```
export function increaseFontSize () {
  return (dispatch, getState) => {
    dispatch({
      type: UI_FONT_SIZE_INCR,
      effect () {
        const state = getState();
        const old = state.ui.fontSizeOverride || state.ui.fontSize;
        const value = old + 1;
        dispatch({
          type: UI_FONT_SIZE_SET,
          value
        });
      }
    });
    };
  }
```

The underlying terminal

`Hyper` achieves a lot of its speed and functionality thanks to the power of hterm underneath, the terminal emulator of the Chromium project.

To access the terminal object, you can supply a `onTerminal` property to the `<Term>` component.

Additional APIs

The Electron `app` objects are extended with the following properties:

`config`	An `Object` with the `config` block from `~/.hyper.js`.
`plugins`	An `Object` with helpers for plugins.
`getWindows`	A `Function` that returns an `Set` of all the open windows.
`createWindow`	A `Function` that will create a new window. Accepts an optional `callback` that will be passed as the new window's `init` callback.

Electron `BrowserWindow` objects are extended with the following parameters:

`rpc`	An `EventEmitter` that allows for communication with the window process.
`sessions`	A `Map` of `Session` objects which hold the communication with each term's pty..

Renderer windows are similarly extended with:

`rpc`	An `EventEmitter` that allows for communication with the window process.
`store`	The Redux `Store` object. This allows access to `dispatch` actions or read the global state wi `getState`.

The `rpc` object is symmetrical between browser and renderer process. The API is the same as Node.js, with the exception that it only admits a single object as its parameters only:

```
win.rpc.emit('hi there', {
  some: 'payload',
  any: [
    'object',
    'here'
```

```
      ]
    });
```

Example theme: Hyperyellow

The following extension simply alters the config to add CSS and yellow colors! Here's the code.

```
▲ ~ vi .h
```

Themes are simply plugins! Only one hook, `decorateConfig` is needed:

```
exports.decorateConfig = (config) => {
  return Object.assign({}, config, {
    borderColor: 'yellow',
    cursorColor: 'yellow',
    css: `
      ${config.css || ''}
      .tabs_nav .tabs_list .tab_text {
        color: yellow;
      }
      .tabs_nav .tabs_title {
        color: yellow;
      }
    `
  });
}
```

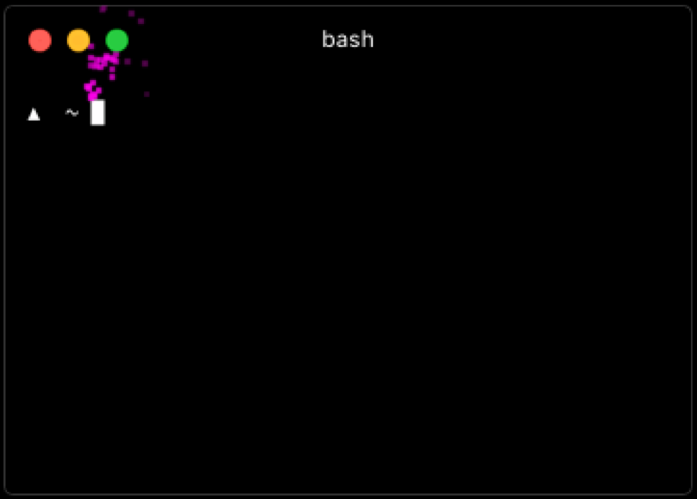
I grabbed the class names by inspecting the term with Devtools, which you can trigger from `View -> Toggle Developer Tools`. When you do so, notice that some classes are automatically generated and followed by a random nonce (e.g.: `term_13hv8io`). Ignore those: they change with every new window!

Notice the emphasis on playing nice with other extensions. Specificaly, we create a new object, extend only the keys we are interested in, and we compose the CSS to preserve the user's setting and that of other authors':

```
    return Object.assign({}, config, {
      css: `
        ${config.css || ''}
        /* your css here */
      `
    });
```

Example extension: Hyperpower

The following extension renders particles as the caret moves:



Let's walk through its code.
First, we intercept the Redux action `SESSION_ADD_DATA`. See the whole list of them here.

```
    exports.middleware = (store) => (next) => (action) => {
      if ('SESSION_ADD_DATA' === action.type) {
        const { data } = action;
        if (/bash: wow: command not found/.test(data)) {
          store.dispatch({
            type: 'WOW_MODE_TOGGLE'
          });
        } else {
          next(action);
        }
      } else {
        next(action);
      }
    };
```

Notice that we don't re-dispatch the action, which means we never render the output of the command to the terminal. Instead, we dispatch an action of our own, which we grab in the `uiReducer` and later map:

```
exports.reduceUI = (state, action) => {
  switch (action.type) {
    case 'WOW_MODE_TOGGLE':
      return state.set('wowMode', !state.wowMode);
  }
  return state;
};

exports.mapTermsState = (state, map) => {
  return Object.assign(map, {
    wowMode: state.ui.wowMode
  });
};
```

We then want to decorate the `<Term>` component so that we can access the underlying caret.

However, `<Term>` is not a container that we can map props to. So we use `getTermProps` to pass the property further down:

```
exports.getTermProps = (uid, parentProps, props) => {
  return Object.assign(props, {
    wowMode: parentProps.wowMode
  });
}
```

The extension then returns a higher order component to wrap `<Term>`. Notice we pass the `onTerminal` property to access the underlying hterm object:

```
render () {
  return React.createElement(Term, Object.assign({}, this.props, {
    onTerminal: this._onTerminal
  }));
}
```

Credits

Authored by **Guillermo Rauch** - @rauchg.
Brought to you by ▲ZEIT. Hosted on now.

Special thanks to the following people:

- Jeff Haynies for his work on polish for general terminal behavior.
- Nuno Campos for his work on zooming and configuration.
- Leo Lamprecht and Johan Brook for their excellent UI improvements.
- Harrison Harnisch for our nice default color palette.
- Fernando Montoya for his feedback and patches.
- Matias Tucci for his work on the auto updater.
- Sebastian Markbage for his insight on the higher-order component extensibility API.

- Joel Besada for his editor particles idea and Zero Cho for his reference implementation.